

Java™ magazine

By and for the Java community 



16

GAME ON!

For two developers, Java ME is the key to winning in India's mobile games market

31

OPENJDK: THE INTERVIEW

Oracle's Donald Smith on *the* project for Java SE collaboration

51

USING PROPERTIES AND BINDING IN JAVAFX 2.0

Oracle's Jim Weaver shows you how to implement the power of binding

DEVICE CONVERGENCE

Jargon Technologies creates unique interactive and immersive experiences with Java 12


```
//from the editor /
```



What

What's not to love about crowdsourcing? Without a clear and obvious topic in mind for this with a follow-on case of writer's block—I went to you, the community, via Twitter for suggestions. Several of them, but the desire for a status update about JDK 8 seemed to be a common theme.

As previously [announced](#), the GA date for JDK 8 is currently expected to be September 2013. The main drivers are, of course, the [Lambda](#) and [Jigsaw](#) projects. Each of these projects could arguably represent one of the most significant and anticipated changes ever to the Java platform, to say nothing of two of them.

The curiosity about JDK status offers a welcome opportunity for us to deep-dive into the OpenJDK project overall—for example, to explore its expanding role in the community ecosystem and its relationship to the Oracle JDK. In this issue, we’ve banked on that opportunity via [an interview with Donald Smith](#), a member of Oracle’s Java SE team who helps steward the OpenJDK community. As you’ll learn, one of the project’s main objectives—and great successes thus far—is to bring a diversity of viewpoints into the development process.

And speaking of diverse viewpoints: before you turn the page, I want to thank you for helping *Java Magazine* break the 100,000-subscriber barrier after only a handful of issues. Without your interest and support, this publication would not, and could not, exist. Do you see a pattern here?

Justin Kestelyn, Editor in Chief BIO



P.S. Starting in this issue, the “Java Champion Duke” next to a byline signifies the author’s membership in that august community. They deserve it!

PHOTOGRAPH BY BOB ADLER



GIVE BACK! ADOPT A JSR

Find your JSR here



//send us your feedback /

We'll review all suggestions for future improvements. Depending on volume, some messages may not get a direct reply.





LEARN WITHOUT LIMITS

20,000+ Books & Videos. One Monthly Price.

Subscribe to Safari Books Online and gain immediate, unlimited access to hundreds of the most important Java books and training videos. Learn about any of the trending technology and business topics from the world's most trusted publishers.



Sign up for a free trial today and save!

safaribooksonline.com/javamag

Access Safari Books Online on virtually any device with a browser:





JAVA TECH

ABOUT US

○

f

ava
net

olog





MAX BONBHEL PHOTOGRAPH BY
ALLEN MCINNIS/GETTY IMAGES

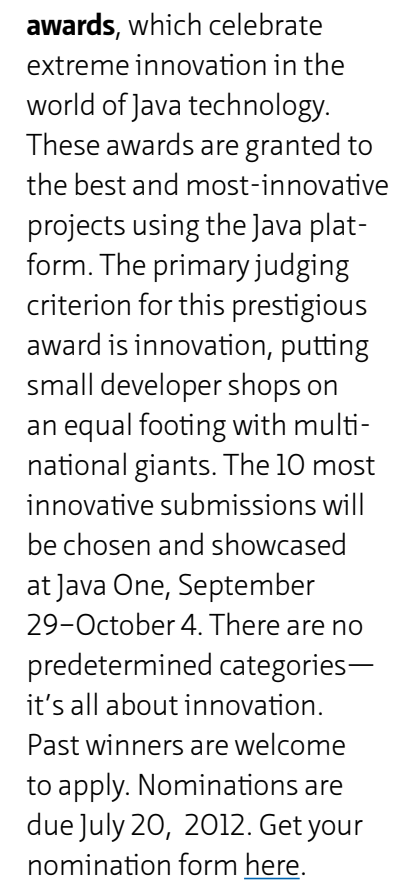


Swedish video gaming company Mojang is encouraging gamers to go beyond just playing its games by learning game programming. Jens Bergensten, lead designer and developer for Mojang's Minecraft game, created a [getting-started guide](#) to programming for beginners using the source code from "Catacomb Snatch" and the Eclipse integrated development environment. "Catacomb Snatch" is a popular 2-D shooting strategy game. The game was part of a Humble Bundle Mojam, a 60-hour charity programming marathon from which half a million dollars in proceeds went to charity. In addition to playing with "Catacomb Snatch" source code, gamers have access to Java 2-D and 3-D [gaming resources](#). They can also share their games and enter competitions such as the one-hour "[Let's Get Petite](#)" competition.

Following the March 2012 death of 200 Congolese citizens due to an explosion in Brazzaville, Congo, **Lamine Ba**, the leader of the Java user group (JUG) in Senegal, used Java to create an [online payment Website](#) to accept donations for the victims. Congo and Senegal are 2 of the 18 countries that are affiliated with [JUG-Africa](#). This umbrella organization was created by **Max Bonbhel** in 2010 to share Java expertise among professional developers. For example, the community organized a dozen Java 7 JUG meetings in the summer of 2011. Ba hopes that JUG-Africa will be the network that will change the continent. "If we can create an impact through this project, then we can become a major force when it comes to supporting the too-frequent crises in Africa," he says.



Scenes from JUG
Chennai, left to right:
a speaker captivates
the group, holding
a Java Summit,
celebrating the
launch of Java 7



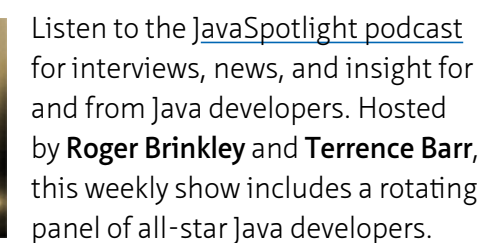


Round Two winners will travel to Astana, Kazakhstan in late April to compete in Round Three, the 2011–2012 Java Olympics championship finals.

Weaver and Chin Join Oracle



JavaSpotlight Podcast



A woman with long dark hair, wearing a dark pinstriped blazer over a white button-down shirt, is speaking at a podium. The podium is orange and has the word "ORACLE" in white capital letters. A microphone is positioned in front of her. The background is dark, and a portion of a screen to the left shows the word "ORACLE" in red.

A portrait of a man with long dark hair and glasses, wearing a dark jacket, speaking into a microphone. This is likely the author, Shigeo Fukuda, mentioned in the text.

A major focus of the conference was the Java roadmap. At the Java Strategy keynote on the first day of JavaOne Tokyo, Oracle's Java engineering executives **Cameron Purdy**, **Nandini Ramani**, and **Henrik Stahl** stepped through Oracle's technology roadmap for Java SE, JavaFX, Java ME/embedded Java, and Java EE. The key takeaways: Java SE 8 will include [Project Jigsaw](#) and [Project Lambda](#); Java ME/embedded Java will synchronize CLDC and JDK releases; and Java EE 7 will support elasticity for capacity on demand and multitenancy. **Sharat Chander**, group director of Java technology outreach at Oracle, also called on local Japanese developers to continue their active participation in the Java community by joining Oracle Technology Network.



MICHAEL HÜTTERMANN: Up Close and Personal

Hüttermann: I'm a team player, and I've always very

Hüttermann: Meeting many

You can read more about Michael Hüttermann in the recent article "[Agile ALM: A Conversation with Java Champion and ALM Expert Michael Hüttermann.](#)" You can also visit [his blog](#) and find him on Twitter (@huettermann).



Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 275 events with more than 40,000 attendees. NFJS is known for knowledgeable speakers and timely presentations that cover the latest trends within the Java ecosystem and agility space. Upcoming symposium dates include the following:

MAY 18-20
DALLAS, TEXAS

JUNE 8-10
COLUMBUS, OHIO

JUNE 29-30
SALT LAKE CITY, UTAH

JULY 13-14
COLUMBIA, MARYLAND

MAY 16-18
POZNAŃ, POLAND

GeeCON focuses on Java- and Java Virtual Machine-based technologies, with special attention paid to dynamic languages such as Groovy and Ruby.

MAY 19-20
SAN MATEO, CALIFORNIA

See how Java will help build the future at this festival of invention, creativity, and resourcefulness.

MAY 21-25
COPENHAGEN, DENMARK

These conferences are designed for software developers, IT architects, and project managers in the Java, mobile, .Net, open source, Lean/agile, architecture, new languages, and process communities.

JUNE 19-22
DENVER, COLORADO
Explore the evolving
ecosystem of the Java

JUNE 26-28
ZURICH, SWITZERLAND
Jazoon, the international
conference on the modern
art of software, will cover a
broad range of topics and
cater to a wide audience
of professionals involved
in software development.
The main track covers
programming and markup
languages; frameworks,
tools, platforms, and
methods; and more.

The Developers
Conference 2012

JULY 4-8
SÃO PAULO, BRAZIL
One of Brazil's largest developer conferences, this conference offers 35 tracks, several of which are Java related.

JULY 5
STUTTGART, GERMANY

The Java Forum offers participants the opportunity to inform themselves about topics related to Java and the Java environment. The forum includes basic lectures, presentations, and information about and demos of specific products.

*JULY 25–28
PORT ALEGRE, BRAZIL*

Sponsored by the Fórum Internacional do Software Livre (FISL), this event is one of the world's largest free software events, and includes technical, political, and social discussions about open source software.

SANTA CLARA, CALIFORNIA
The 2012 JVM Language Summit is an open technical collaboration between language designers, compiler writers, tool builders, runtime engineers, and virtual machine architects.

OSCON JULY 16-20, PORTLAND, OREGON

Join the world's open source pioneers, builders, and innovators for five days of sessions and keynotes on open source languages, platforms, and development. OSCON features more than 200 speakers, hundreds of technologies, and 3,000 hackers. Tracks include Java and JVM Languages, Geek Lifestyle, JavaScript and HTML5, Mobile, Tools and Technologies, and much more.

ORACLE.COM/JAVAMAGAZINE ////////////////////////////////////// MAY/JUNE 2012

JAVA IN ACTION

JAVA TECH

ABOUT US

0



Pradyumn Kumar

Expert Oracle and Java Security

Programming Secure Oracle Database Applications With Java

Second Edition

Apress®

By Heiko Böck
Apress (December 2011)
The Definitive Guide to NetBeans Platform 7 is a thorough and authoritative introduction to the NetBeans platform. The book provides a completely updated definitive guide to the NetBeans platform, using the latest APIs, coding patterns, and methodologies. It focuses strongly on business features in an application, with the author covering how to use OSGi, how to add authentication/security, and how to monetize from a modular application. You'll learn how to get started using the NetBeans platform with or without using the NetBeans IDE, how to set up a modular application, and more.

By Stuart Halloway and Aaron Bedra
Pragmatic Bookshelf
(April 2012)
Programming Clojure, Second Edition is a significant update to the classic book on the Clojure language. You'll get thorough coverage of all the new features of Clojure 1.3, and enjoy reorganized and rewritten chapters that reflect the significance of new Clojure concepts. Many code examples have been rewritten or replaced, and every page has been re-evaluated in light of Clojure 1.3. As the authors show you how to build an application from scratch, they provide a rich view into a complete Clojure workflow. You'll also get an education in thinking in Clojure.

Data Quality Tools for Java



BEFORE

john smith iii phd
melissa data corp.
22382 Empresa 92688
7145895200
john@800miAL.con

AFTER

Melissa Data Corp.
John Smith III PhD
22382 Avenida Empresa Ste 100
Rancho Santa Margarita, CA 92688-2112
949-589-5200
John@melissadata.com
Delivery Indicator: Business

*Highlights indicate added and/or corrected data.

Realtime NCOA^{Link}
Change-of-Address
Web Service
available

Now, finding the right data verification tools doesn't have to be so puzzling. Melissa Data offers customizable APIs, Web services and enterprise applications to match your budget and business needs. For solutions to cleanse, validate and standardize your contact data, we're ready to help you find the perfect fit.



Request free trials at
MelissaData.com/myjava or call 1-800-MELISSA

- Global address verification for 240 countries
- Clean and validate data at point-of-entry or in batch
- Correct misspellings, missing directionals, and confirm deliverability
- Enhance addresses with County, Census, FIPS, etc.
- Append rooftop lat/long coordinates to street addresses
- Update records with USPS and Canadian change of address info

MELISSA DATA[®]

Your Partner in Data Quality

BY DAVID BAUM

PHOTOGRAPHY BY PHIL SALTONSTALL
ART BY I-HUA CHEN



Troy Mockenhaupt,
senior engineer at
Jargon, tests out
various devices.

expertise that was lacking in the industry, and there were only a few companies that were able to do this," she recalls. "We founded Jargon Technologies to realize the vision of

innovative use of Java technology—specifically, to help special-needs children realize their full potential.”

The market reached a tipping point in 2008 when several studios and distributors shifted their allegiance to the Blu-ray format. On February 19, 2008, Toshiba officially announced that it would stop the development of HD DVD players. Blu-ray had won the format wars, and, almost overnight, Jargon found itself at the center of a revolution in interactive entertainment.

"From the outset, we saw this opportunity as much more than just content for Blu-ray players," Srikanth says. "We were thinking about communications, about home networks, and about connected devices. We said, 'Let's see how much Blu-ray can do. . . .'"

SNAPSHOT

**JARGON
TECHNOLOGIES**

jargon-tech.com

Headquarters:

Burbank, California

Industry:

Media and entertainment

Employees:

15

Java version used:

Blu-ray Disc Java, based on
Java ME CDC/PBP Packaged
Media profile of Globally
Executable MHP (GEM)

Srikanth grew up in India, where she also attended college and obtained a master's degree; she then worked in Japan helping an elite group of software engineers automate a steel mill. She transitioned into media technology in 2002 when she joined Panasonic Hollywood Laboratories (PHL) in Southern California, where she was part of the core R&D team that developed the Blu-ray Disc high-definition optical media format. It was an exciting time for the home entertainment industry. Along with Sony and Philips, Panasonic was working diligently to gain market acceptance against the rival HD DVD format backed by Toshiba. Srikanth represented Panasonic at several meetings of the Blu-ray Disc Association and also worked with some of the Hollywood studios to create Blu-ray content.

As consumer interest in high-definition video content increased, Srikanth and her colleagues saw an opportunity to strike out on their own.

"We started seeing a need for Blu-ray

the creative community in Hollywood, as well as in other industries such as education."

BIRDS OF A FEATHER

Jargon's other founding members included Jeff Schulz, chief platform architect, and Nathan Epstein, COO. Srikanth donned the mantle of CEO, taking responsibility for account management, business development, and overall corporate strategy. The new company soon landed high-profile projects with Panasonic, Walt Disney Pictures, Warner Brothers, and Twentieth Century Fox Home Entertainment. Its forte was developing interactive applications that reside on the bonus discs distributed with popular Blu-ray titles.

"Slowly, we started doing more and more Blu-ray work, but we also branched out into connected TV, mobile, and other areas," says Srikanth. "Our goal is to reach a wide audience through

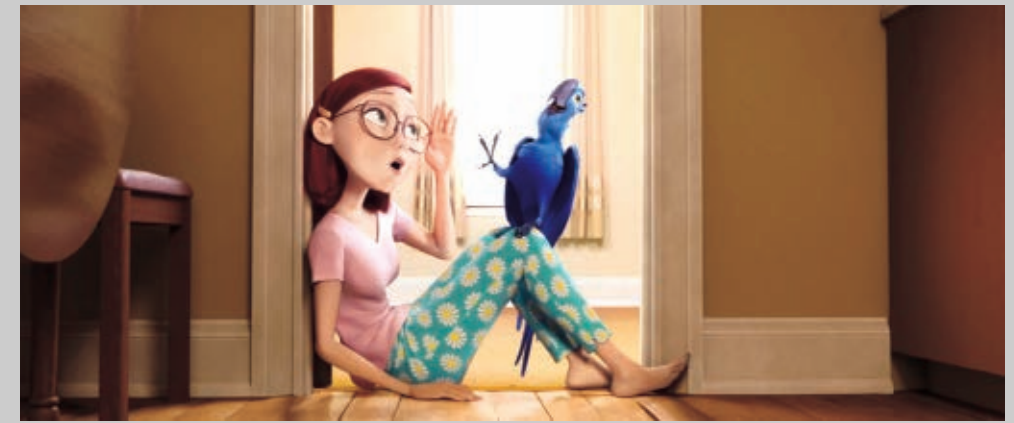
TAKING FLIGHT

Be careful what you wish for. It was around this time that Twentieth Century Fox Home Entertainment asked Jargon to implement a new type of game for the Blu-ray release of the movie *Rio*, a very successful animated film about a domesticated macaw from small-town Minnesota who takes off on an adventure to Rio de Janeiro, Brazil. The theatrical release had been a worldwide success,

with box office earnings nearing US\$500 million dollars. Now it was time to bring the family adventure to the cable and home entertainment market.

Jargon's technology team brainstormed with Fox Home Entertainment's content directors about ways to push the boundaries of Blu-ray Disc Java (BD-J) to enhance the consumer

THE POWER OF BD-J
Blu-ray Disc Java
powers bonus view
content including
network access,
picture-in-picture,
and local storage.



BD-J is a specification for creating advanced content on Blu-ray Discs—it's what makes Blu-ray Disc titles so much more sophisticated than similar content on a standard DVD player, including network connectivity, picture-in-picture (PIP), and access to local memory storage. In this case, it lets *Rio* viewers capture, resize, rotate, and position their

Thanks in part to its **innovative interactive content**, Rio reigned as the #1 Blu-ray on retail charts for four weeks in a row.

BY PHILIP J. GILL

The “smallest” Java platform, Java ME, is playing a major role in helping mobile application developers profit in one of the world’s largest and fastest-growing economies: India. With 1.2 billion people, India is the world’s second-most-populous country, behind China. Not surprisingly, this South Asia giant is also the world’s second-largest mobile

PHOTOGRAPHY BY NAMAS BHOJANI



**Top: Twist Mobile
CEO Virat Khutal;
Bottom: Nextwave
Multimedia CEO P.R.
Rajendran (left) and
COO P.R. Jayshree**

We didn't invent the Internet...

...but our components help **you** power the apps that bring it to business.



PURE JAVA COMPONENTS & ENTERPRISE ADAPTERS FOR

- **E-Business**
AS2, EDI/X12, NAESB, OFTP ...
- **Credit Card Processing**
Authorize.Net, TSYS, FDMS ...
- **Shipping & Tracking**
FedEx, UPS, USPS ...
- **Accounting & Banking**
QuickBooks, OFX ...
- **Internet Business**
Amazon, eBay, PayPal ...
- **Internet Protocols**
FTP, SMTP, IMAP, POP, WebDav ...
- **Secure Connectivity**
SSH, SFTP, SSL, Certificates ...
- **Secure Email**
S/MIME, OpenPGP ...
- **Network Management**
SNMP, MIB, LDAP, Monitoring ...
- **Compression & Encryption**
Zip, Gzip, Jar, AES ...

The Market Leader in Internet Communications, Security, & E-Business Components

Each day, as you click around the Web or use any connected application, chances are that directly or indirectly some bits are flowing through applications that use our components, on a server, on a device, or right on your desktop. It's your code and our code working together to move data, information, and business. We give you the most robust suite of components for adding Internet Communications, Security, and E-Business Connectivity to

any application, on any platform, anywhere, and you do the rest. Since 1994, we have had one goal: to provide the very best connectivity solutions for our professional developer customers. With more than 100,000 developers worldwide using our software and millions of installations in almost every Fortune 500 and Global 2000 company, our business is to connect business, one application at a time.

connectivity
powered by 

To learn more please visit our website →

www.nsoftware.com

The `placeNewPizza()` method will place a new slice into the world every time a slice gets eaten. This method does not exist—we will have to write it now.

To place a new pizza slice at a random location into the world, we have to achieve three things:

- The first line in our new method reads

We have seen before that the `get-World()` call gives us a reference to our current world object. This time, we declare a local variable named `myWorld` (it is of type `World`) and store the world object in it. This, in itself, does not achieve very much yet, apart from the fact that we can now use the `myWorld` variable to talk to our world.

In the next two lines, we generate the random coordinates and store them in two local integer variables:

We have seen method calls to the `getRandomNumber`, `getWidth`, and `getHeight` methods before. Here, we generate a random number between 0 and the world's width and store it in the `x` variable. Then we do the same for `y` using the height of the world as the limit. This gives us random `x` and `y` coordinates that match the exact size of the world.

```
myWorld.addObject(new Pizza(), x,  
y);
```

Here, we place the new pizza object into the world. To do this, we are using the world's `addObject` method. This method expects three parameters: the object to be placed, the x coordinate, and the y coordinate.

We can write `new Pizza()` to create a new object of type `Pizza`, and we can use this in place of the object to be placed into the world. Then we can use our `x` and `y` variables (which we have just filled with our random values) in place of the coordinates.

Read the complete **Listing 1** again and make sure that you understand it. Then try it out for yourself.

The functionality so far gives us an endless supply of pizza slices. The next task to be done is to count how much we have eaten. We do this by declaring a *field* (also known as an *instance variable*) named `slices` to use for the counting. (We're still working in the `Turtle` class.)

```

/**
 * Look for pizza and eat it if we see some.
 */
public void eat()
{
    Actor pizza = getOneIntersectingObject(Pizza.class);
    if(pizza != null) {
        getWorld().removeObject(pizza);
        Greenfoot.playSound("slurp.wav");
        placeNewPizza();
    }
}

/**
 * Place a new pizza slice at a random location into the world.
 */
public void placeNewPizza()
{
    World myWorld = getWorld();
    int x = Greenfoot.getRandomNumber(myWorld.getWidth());
    int y = Greenfoot.getRandomNumber(myWorld.getHeight());
    myWorld.addObject(new Pizza(), x, y);
}

```


Download all listings in this issue as text

This is what it looks like:

The field declaration consists of the keyword `private`, followed by a type and name for our variable, and an optional initialization value. The type and name (`int slices`) is the same as we have seen

for a local variable. The initialization (`= 0`) assigns zero to the variable to start.

The placement of this line is important. Field declarations must be outside of method definitions, so we write this line immediately after the class header (before our `act` method).

The `slices` field can store integers (that is, whole numbers). It currently holds zero. We can now use this field to count our pizza slices. Every time we eat some pizza, we increment this field by 1. When we have reached 12, we play the fanfare (the sound file `fanfare.wav`, which is

included in the project download) and stop the game.

Listing 2 shows the complete implementation. Let's briefly examine the interesting aspects.

We have added the following line:

- slices++;

The `++` operator, written after an `int` variable, increments (that is, counts up) this variable by 1. So we can see that we are counting up every time after removing a pizza slice from the world.

Next, we have the following lines (with some instructions in the place of the three dots):

```
if(slices == 12) {
    ...
}
```

```
}
else {
    ...
}
```

This code checks whether the `slices` variable is now equal to 12 and, if it is, executes the instructions between the first pair of curly brackets. Otherwise (if the variable is not 12), the instructions after the `else` keyword are executed. Note that we need to use a double equal symbol (`==`) to check for equality.

Read **Listing 2** carefully. You will see that this code is used to play our slurping sound and place a new pizza slice as long as we have not eaten 12 slices yet. Then it plays the fanfare and ends the game when we have eaten 12 slices. Try the code out in your own version of the game.

Sharing Your Game

Now that our game is “complete” in some sense—of course, no game is ever truly finished—it would be great if we could get all our friends to try it out. Luckily, we can.

Greenfoot has a “share” function that allows you to upload your program to the Greenfoot Website, where other users can play it and give you some feedback. You can see all the scenarios other Greenfoot users have created by going to the Greenfoot site and looking around.

To upload your own game, click the **Share** button in the top right of Greenfoot's main win-

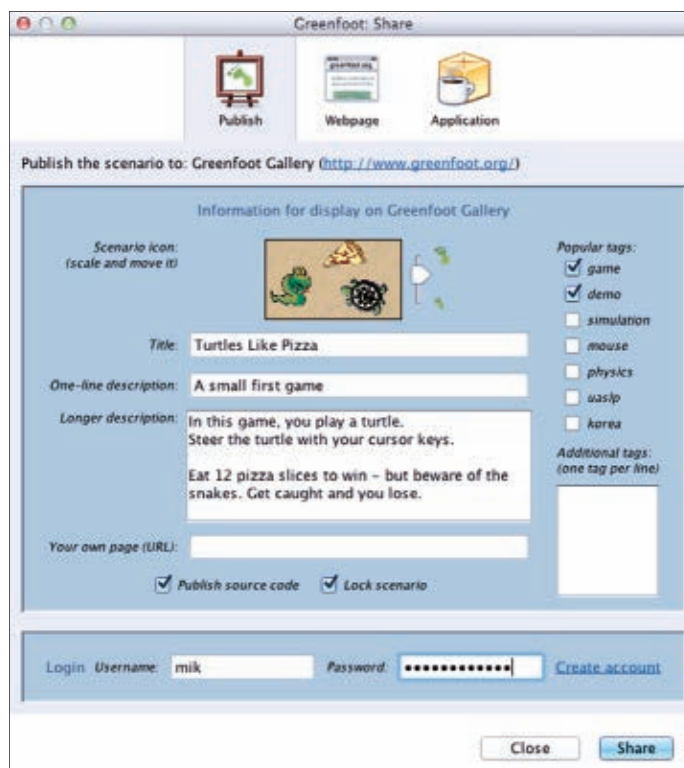


Figure 2

LISTING 2

```

/**
 * Look for pizza and eat it if we see some.
 */
public void eat()
{
    Actor pizza = getOneIntersectingObject(Pizza.class);
    if(pizza != null) {
        getWorld().removeObject(pizza);
        slices++;
        if(slices == 12) {
            Greenfoot.playSound("fanfare.wav");
            Greenfoot.stop();
        }
        else {
            Greenfoot.playSound("slurp.wav");
            placeNewPizza();
        }
    }
}
}

```



[Download all listings in this issue as text](#)

dow. Clicking it brings up a dialog box that allows you to specify some details for your upload (see **Figure 2**).

For the upload to work, you must have an account on the Greenfoot site. (If you don't, click the **Create account** link.) You can browse and play other people's games without signing up, but to upload your own game, you need an account.

Once you have an account, this dialog box lets you add a title, a description, an icon, and some tags, and then you can upload your game. When you upload a game, you can choose whether you want to upload the source code, too. If you do, others can look at your source code and possibly learn from it or modify it themselves.

Your project will have a unique URL on the Greenfoot Website. Send this to your friends to show them the cool things you have done!

Conclusion

With this article, we have finished building a simple computer game using Greenfoot and Java. If you followed along throughout this series, you will have seen that building a simple game in Java is not very hard. And you can learn important programming concepts along the way. Now take a look around the Greenfoot site for some more ideas, and continue to build your own games. And above all, have fun! `</article>`



R GINA TEN
BRUGGENCATE
AND **LINDA** VAN DER PAL

Build a Web App in Wicket 1.4 Using NetBeans

Combine Wicket and Java EE 6 to build a Web app easily by separating markup from business logic.

In this tutorial, we will show you how to build a simple Web application in [Apache Wicket 1.4](#) using the NetBeans IDE. Wicket is a component-based Web application framework that lets you build enterprise-grade Web applications using Plain Old Java Objects (POJOs), HTML, Ajax, Spring, Hibernate, and Maven.

We chose Wicket as our subject because we love its separation of markup and logic and the way you can build components that you can reuse. We also wanted

to show how it can be combined with Java EE 6, because Java EE 6 is gaining more and more momentum and is quite easy to learn. And we used NetBeans because it offers good integration between the two.

We would have liked to show you how to build an entire Web shop with Wicket. But you'll just have to read *Wicket in Action* for that. All we have space for in this article is to show you how to display a list of products with a link to a details page. We'll build the

Web app gradually and show you some alternatives along the way.

Note: The source code and other necessary files for the project shown in this tutorial can be downloaded [here](#) (note: 14.6 MB).

Setting Up Your Project

The first part of this tutorial is based on Jeff Schwartz' [blog about Java EE 6 and Wicket](#). If our summary is too concise for your purposes, read the series of articles mentioned in Schwartz' blog.

Downloading the required software and files.

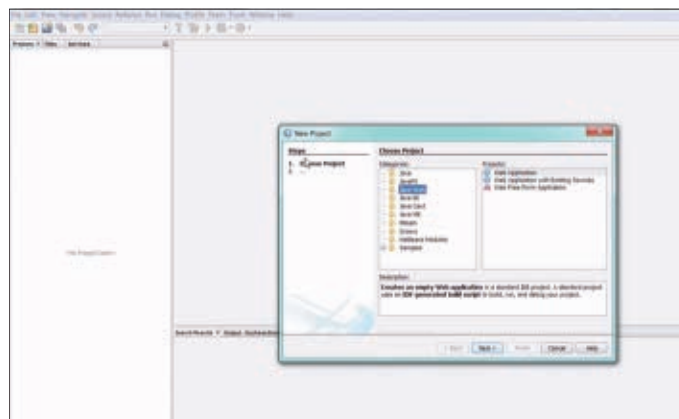
In this section, you are going to set up your development environment.

1. Download and install the following software: [MySQL](#), [NetBeans](#), and the [Wicket plug-in for NetBeans](#).
2. You will use `create_script_mysql.sql` to create the backing database. Execute this script in the command line tool by running the following line:
`\\Create_script_mysql.sql`.

Creating the project. In this section, you will create the project and create the default Wicket application that will be modified in a later section.

1. In the NetBeans IDE, from the **File** menu, select **New Project**.
2. Select **Java Web** in the Categories panel, select **Web Application** in the Projects panel, and click **Next**.
3. Enter **DuchessStore** in the **Project Name** field, and click **Next**.
4. Make sure **GlassFish Server 3** is selected as the server and **Java EE 6 Web** is selected as the Java EE version, and click **Next**.
5. In the Frameworks panel, select **Wicket**. (If Wicket isn't available as a framework, the Wicket plug-in wasn't installed correctly.)
6. Enter **org.duchess.example.wicket** for **Main Package**, and click **Finish**.

Note: To see these steps in action, view the [Set Up Project](#) movie.



Watch the Set Up Project movie to see the steps to create the project.

ORACLE.COM/JAVAMAGAZINE //////////////////////////////////// MAY/JUNE 2012

`BasePage` and we are no longer passing a `String` to the `HeaderPanel`. Now replace the code with the new code, as shown in **Listing 3**. This will cause a compiler error, which we will fix in the next two steps.

3. In the [org.duchess.example.wicket](#) package, you can find the `HeaderPanel.html` file that needs to be replaced with the code shown in **Listing 4**. We have added a stylesheet, as well as an image with our logo, to the [HeaderPanel](#). We have also changed the text that is shown. This will show on the top of all the pages that inherit from the [BasePage](#).
4. And finally the code in the `HeaderPanel.java` file will be replaced with the code found in **Listing 5**. This will resolve the compiler error from Step 2.

Now we come to the real work. In order to show all the products from our database, we will use the simplest component Wicket offers to make a list of repeating items: the `RepeatingView`. This component renders all its children in the order in which they were returned from the database.

5. In the `org.duchess.example.wicket` package, you can find the `HomePage`

.html file. This is the first page that is shown when the application is started. Replace the code with the code from **Listing 6**. We added a table to the page. We gave it some headers and a single row. This row has a **wicket-id** that we will use to couple this row to the **RepeatingView**. The row has three columns that contain the data we want to show.


6. In the same package, you can find the `HomePage.java` file. This code should be replaced with the code from **Listing 7**. In the `HomePage` class, we create a `RepeatingView` instance and give it the same ID as our row. We add this repeater to our page. Then we fetch all our products from the database and loop over them.

For each item, we create a

WebMarkupContainer. As explained in *Wicket in Action*, **WebMarkupContainer** is a generic component that in itself doesn't do much. It can contain child components, so it is a handy tool to group components or use as an intermediate layer when you need to group more markup for a component. If we didn't use it here, we would get an exception when trying to add more than one product to our page, because the ID **name** is

used more than once. If we run the application, the page now looks like **Figure 1**.

Adding paging. If our database ever grows to contain more products than would fit in a single screen, our solution



Java EE6 and Wicket EJB Tutorial

Name	Description	Price
Buttons 4x4x4	Small Duckies button	1
Buttons 4x4x4	Large Duckies button	400
Blue shirt L	Blue Duckies on Your polo size L	2000
Blue shirt XL	Blue Duckies on Your polo size XL	4000
Hoodie	White Duckies hoodie	2000
Socks	Socks with Duckies logo	7000
Knockout	White hoodie with Duckies logo	350
Puppet Kill	Make your own Duckies Puppet Kill	1200

Figure 1

LISTING 4

LISTING 5

LISTING 6

LISTING 7

LISTING 8

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns:wicket="http://wicket.apache.org">
  <head>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <wicket:panel>
      <wicket:link>
        
      </wicket:link>
      <h1>
        <span wicket:id="headerpaneltext"/>
      </h1>
      <hr/>
    </wicket:panel>
  </body>
</html>
```



Download all listings in this issue as text

wouldn't look very nice anymore. We would like to add paging to solve that. In order to do this, we will change the `RepeatingView` into a `PageableView`.

1. We will first change the `RepeatingView` into a `ListView`. For this, only the `HomePage.java` file needs to be changed. The HTML file stays the

same. As you can see in **Listing 8**, this code is a little bit simpler. We don't need to loop over the items ourselves anymore; the **ListView** does that for us. And we no longer need the **WebMarkupContainer**. All we have to do is override and implement the **populateItem** method. Now we replace

the code in the `HomePage` with the code shown in **Listing 8**.

The next step is to add actual paging. Once again, this is a relatively small step. But this time we need to change both the HomePage.java file (shown in **Listing 9**) and the HomePage.html file (shown in **Listing 10**).

2. As you can see in **Listing 9**, all we needed to do was change the type of the **ListView** and add a parameter to the constructor indicating how many elements we want to show in the list. We also needed to add a **PagingNavigator**, which will handle the navigation through our pages. So now we replace the code in the `HomePage.java` file with the code in **Listing 9**.
3. As you can see in **Listing 10**, the changes in the HTML file are even more minimal; we only needed to add the navigator markup. Now we add the highlighted code from **Listing 10** to the `HomePage.html` file. Now our page looks like **Figure 2**.

When our list grows so large that we need many pages to show our products, this navigator doesn't scale very nicely. It shows at most ten pages and then doesn't inform us about how many more



Figure 2

pages there are. It would be nice if we could show a different navigator when there are many pages and not show a navigator when there is only one page. These changes can be accomplished by creating a **PagerFactory** that will give us different navigators depending on the number of products. We'll also need to create our own navigators and some properties for those navigators.

4. First, create a new `org.duchess.example.wicket.components.navigators` package for the new navigator classes.
5. The first class you need to create is `PagerFactory`. The code for this class can be found in **Listing 11**. This code will cause several compiler errors, but those will be fixed later on.

The next steps will be for the pager that will be shown when there are more than five pages to navigate through.

6. Create a new Wicket Page named `HighNumberNavigator`.
7. Replace the code of the newly created HTML file with the code from **Listing 12**.
8. Replace the code of the newly created Java file with the code from **Listing 13**. You'll have to organize the imports and reformat the code

to make it more readable, because we were constrained by formatting rules.

The next steps will be for the pager that will be shown when there are less than five pages to navigate through.

LISTING 9

LISTING 10

LISTING 11

LISTING 12

LISTING 13


```
package org.duchess.example.wicket;

import java.util.List;
import javax.ejb.EJB;
import org.apache.wicket.markup.html.basic.Label;
import org.apache.wicket.markup.html.list.ListItem;
import org.apache.wicket.markup.html.list.PageableListView;
import org.apache.wicket.markup.html.navigation.paging.PagingNavigator;
import org.duchess.example.beans.ProductFacade;
import org.duchess.example.entities.Product;

public class HomePage extends BasePage {

    @EJB(name = "ProductFacade")
    private ProductFacade productFacade;

    public HomePage() {
        List<Product> products = productFacade.findAll();
        PageableListView<Product> productList =
            new PageableListView<Product>("productList", products, 5) {
            @Override
            protected void populateItem(ListItem<Product> item) {
                Product product = item.getModelObject();
                // name
                item.add(new Label("name", product.getName()));
                // description
                item.add(new Label("description", product.getDescription()));
                // price
                item.add(new Label("price", product.getPrice().toString()));
            }
        };
        add(productList);
        add(new PagingNavigator("navigator", productList));
    }
}
```

 [Download all listings in this issue as text](#)

- [illegible]



MAY/JUNE 2012

Wicket is a **component-based Web application framework** that lets you build enterprise-grade Web applications easily.

When the list of products gets too large, using a list as the model becomes burdensome. Even the `PageableListView` loads the whole list in memory before rendering the items on the page.

interface between the database and the data view.

Our first step for implementing a **DataView** is to create a data provider.

- in **Listing 20**, and organize the imports.

LISTING 14 / LISTING 15 / LISTING 16 / LISTING 17 / LISTING 18 / LISTING 19



[Download all listings in this issue as text](#)

field, and click **Finish**.

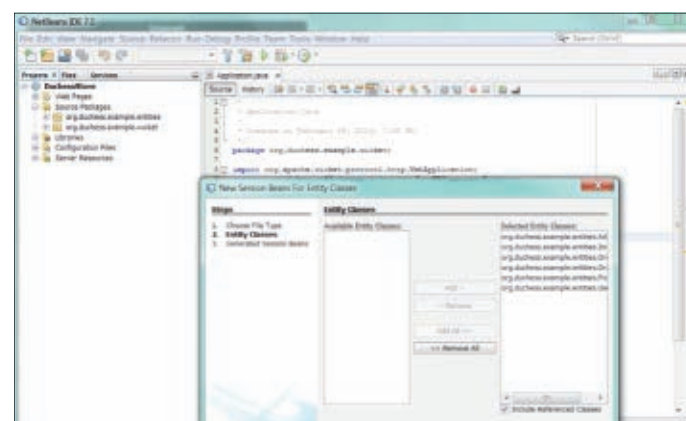
9. Fill the `ProductsPanel.html` file with the code shown in **Listing 21**.
10. Fill the `ProductsPanel.java` file with the code shown in **Listing 22**.
11. And finally, add the new `Panel` to the `HomePage` by changing both the `HomePage.java` file (**Listing 23**) and the `HomePage.html` file (**Listing 24**).

Creating a second page to show the details of a product. Next, we want to create a page where we can show the details of a product. First, we need to create a [ProductDetailPage](#):

-
- The screenshot shows the IntelliJ IDEA IDE with the 'New' dialog box open. The 'Name' field contains 'NewFile.java'. The 'File Type' is set to 'Java File'. The 'Location' is set to 'src/main/java'. The 'Create File Type' button is highlighted. The background shows the project structure of a Java application.



Watch the [Create Entities](#) movie to see the steps to create entities.



Watch the Create Session Beans movie to see the steps to create session beans.

LISTING 20 / LISTING 21 / LISTING 22 / LISTING 23 / LISTING 24

```
package org.duchess.example.wicket.dataproviders;

// imports

abstract public class ProductDataProvider implements IEjbDataProvider<Product> {

    @Override
    public Iterator<? extends Product> iterator(int first, int count) {
        // to make sure we are not going to get a negative number of values
        // to retrieve, we need to reset the count
        if (count < first) {
            count += first;
        }
        int[] range = {first, count};
        List<Product> products = getFacade().findRange(range);
        return products.iterator();
    }

    @Override
    public IModel<Product> model(final Product object) {
        final Long id = object.getId();
        LoadableDetachableModel<Product> ldm = new
            LoadableDetachableModel<Product>(object) {
            @Override
            protected Product load() {
                return getFacade().find(id);
            }
        };
        return ldm;
    }

    @Override
    public int size() {
        return getFacade().count();
    }

    @Override
    public void detach() {
    }
}
```

 [Download all listings in this issue as text](#)

- Apache Wicket

 [Download all listings in this issue as text](#)



Get Collaborative

Oracle's **Donald Smith** discusses the OpenJDK community, the place to collaborate on Java SE. **BY JANICE J. HEISS**

When he returned to Oracle in May of 2011, Smith remarked in his blog about his new job, “The team I’m on has one simple mandate—keep Java the number one computing platform in the world.”

Smith: It was first announced in November 2006, and in 2007 the actual project was created, so people could access the code. From 2007 to 2009, a lot of formation took place as more committers joined. And then, in 2009, Oracle began the process of acquiring Sun, which officially closed in January 2010, and that gave Oracle a chance to reveal its commitment to OpenJDK and ongoing plans.

COMMUNITY

JAVA IN ACTION

JAVA TECH

ABOUT US

ava
net

log





Left to right: Oracle's Donald Smith, director of product management, and Dalibor Topic, principal product manager, OpenJDK, make plans for an OpenJDK community event. Smith and Cecilia Borg, OpenJDK onboarding program manager, talk about recognizing achievements in the OpenJDK community.



Smith: The OpenJDK community has access—through a TCK license agreement specifically for the OpenJDK community—to a compatibility test kit to validate that their builds of OpenJDK are compatible with the Java SE specification. This license is free of charge, but you have to sign the agreement, and we have to process it, which can take a few weeks. You can find the agreement on the OpenJDK Website. We’re trying to keep Java compatible.

Java Magazine: What's happening now in terms of industry participation in the OpenJFX community?

Smith: We announced at 2011 JavaOne that we were going to begin the process of open sourcing JavaFX, and we've made strides with initial code contributions, and now we're looking to ramp up community participation. Several organizations are making strategic use of JavaFX, and we're hoping to see more participation in the project. We anticipate having some exciting announcements related to JavaFX at JavaOne this year.

Java Magazine: Tell us about the various groups who are a part of OpenJDK.

Smith: The notion of a *group* is codified in the OpenJDK bylaws. A group is a collection of participants interested in engaging in an open conversation about a common interest. There are, for example, groups that focus on the core libraries, the compiler, and security. One of the more topical areas right now that my team is focusing on is the quality group, where we want to make it easier for people to submit and run their own test cases and test harnesses. There is a group related to build. The challenge with both of these groups, and a number of our activities, is that we are trying to push the infrastructure along to keep up with the demands of the community. We're working on a better bug reporting system, but we also need a better build and test story so that people can more easily get their own builds up and running.

Luckily, we are getting help. The London Java Community has done a great job of pushing along information about how to do builds and providing very constructive and timely feedback on these topics. And there's a developer from SAP named Volker Simonis who has done a fantastic job at documenting his own experiences building OpenJDK on a number of platforms. There are lots of stars in the community, but we're always looking for more!

Java Magazine: OpenJDK contributors now include such companies as Apple, and as you mentioned, IBM and Twitter. It's noteworthy that Twitter joined, even though they are not a company that gets their revenue from software licensing—they're an end user of Java.

Smith: This is a very important point. When I was at Eclipse, we noticed something that academics and others have written about. There are different waves of participation in open source communities. The first wave is

usually one or two organizations that have a significant code base that, for whatever reason, they're interested in making open source. They are usually ISVs [independent software vendors], or other enterprise software companies, that make their revenues directly from software licensing.

SOLID FUTURE
I believe we've established Oracle's fundamental commitment to Java through OpenJDK. We have a roadmap for releases, a Java Virtual Machine strategy for HotSpot and Oracle JRockit, and many key industry participants.



Enhance your applications in new and interesting ways.

I will assume that you are already familiar with the shaped and translucent window concept and API. If not, Oracle has a [great tutorial on the topic](#). The

Working with a per-pixel alpha channel is more powerful than shaped and transparent windows because you can control the transparency of each pixel independently. A shaped window is simply a nonrectangular window. All visible pixels are still 100 percent opaque, but the window might be circular or some other funky shape. Translucent windows set a single transparency value for the entire window, which can be useful if you want to fade the window all at once, but

All modern desktop operating systems draw each window into an offscreen buffer before copying the buffer to the screen. The operating system uses the alpha part of each pixel to decide how to blend the pixel with the rest of the windows on the screen. By having control over the alpha channel, we can make all sorts of interest-

Notice that the gradient is made of four separate colors and each color has an alpha component (the last argument to the

While shaped windows have long been possible using various tricks and hacks, they are now an official part of Java SE 7.

JAVA IN ACTION

JAVA TECH

ABOUT US



Java
.ne

blo



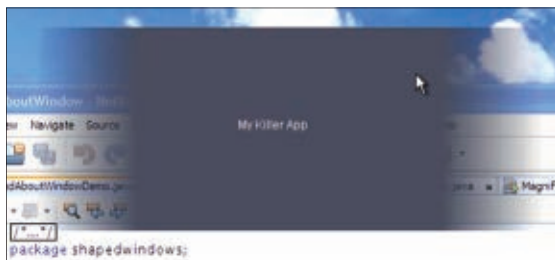


Figure 1

Color constructor). The code in **Listing 1**, by itself, won't actually look transparent onscreen, however. The component will be partially transparent, but all you will see is the background of the window. In Swing, every window has a standard background color (usually some form of gray) on which every component is drawn. To truly make the window transparent, we need a few more things (see **Listing 2**).

The code in **Listing 2** creates the window that the component will be in. It turns off the window decorations and then sets the background to the color 0,0,0,0, which is fully transparent. This effectively turns the background of the window off, letting the desktop background or other applications shine through. Combining this with our custom component, the app looks like **Figure 1**.

The translucent window in **Figure 1** looks great, but there are a few problems. First, because we turned off the window decorations, the user can't move the window. There is no title bar to drag. Second, the user can see

the window in the taskbar, even though we may not want that. It's not enough to just make the window look cool. We have to consider *how the user will actually use it*. Let's look at another example where we solve these problems.

Keystroke Overlay

I often do demonstrations of programs onscreen. The audience can see my mouse cursor but they often don't know what keys I'm pressing, especially if I'm using menu shortcuts, which means nothing shows up onscreen when I press the keys. In these cases, it would be nice to have a window at the bottom of the screen that would show my keystrokes in a nice large font overlaid on top of the app. With a translucent window, it would be a snap.

First, we need to capture the key-strokes. I could put a keyboard listener on every text component in my entire application, but there is an easier way.

Swing lets you add a listener directly to the event queue, which enables you to see all the keystroke events in one place. **Listing 3** shows a code snippet of adding the [AWTEventListener](#).

The code in **Listing 3** adds a listener to the main event queue. It uses a mask to filter out anything that isn't a keystroke. If the keystroke event is a `keyPressed` event (rather than a `keyReleased` or `keyTyped` event), the code will tell the overlay window

LOTS OF OPTIONS

In the old days, we had a fairly small number of UI controls: buttons, sliders, checkboxes, scroll bars, and menus. **Today we have many kinds of custom controls.**

LISTING 1

LISTING 2

LISTING 3

LISTING 4

LISTING 5

```
private static class AboutComponent extends JComponent {
    public void paintComponent(Graphics graphics) {
        Graphics2D g = (Graphics2D) graphics;

        //create a translucent gradient
        Color[] colors = new Color[]{
            new Color(0,0,0,0)
            ,new Color(0.3f,0.3f,0.3f,1f)
            ,new Color(0.3f,0.3f,0.3f,1f)
            ,new Color(0,0,0,0)};
        float[] stops = new float[]{0,0.2f,0.8f,1f};
        LinearGradientPaint paint = new LinearGradientPaint(
            new Point(0,0),
            new Point(500,0),
            stops,colors);
        //fill a rect then paint with text
        g.setPaint(paint);
        g.fillRect(0, 0, 500, 200);
        g.setPaint(Color.WHITE);
        g.drawString("My Killer App", 200, 100);
    }
}
```



[Download all listings in this issue as text](#)

about it. Now let's move on to the overlay window.

The overlay will be a translucent, gray round-cornered rectangle with large white text in the middle. To do this, I created an overlay component that draws everything using Java 2-D code. The keystroke event listener will set the text using the `setText()` method, which will trigger a repaint and draw the key-stroke onscreen. See **Listing 4**.

Now let's put the overlay in a window. Again, we create a window, turn

off decorations, set the background to transparent, and add the component, as shown in **Listing 5**. There's one slight difference, though; can you see it?

Instead of using `JFrame`, I used `JWindow`. `JWindow` is the base class of `JFrame` and behaves very similarly, but with one important difference. A `JFrame` is a full top-level window with decorations, and it will show up in the operating system's native task manager. In the case of Microsoft Windows, this means the taskbar. A `JWindow` is simply a window.

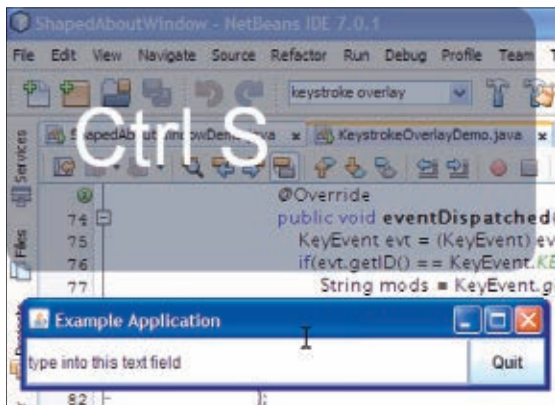


Figure 2

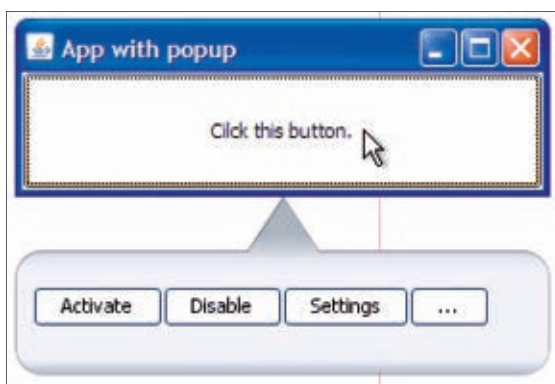


Figure 3

It doesn't represent a large part of your app, so it won't show up in the taskbar and it doesn't have decorations. On Mac OS X, it won't show up in the Expose view either. A **JWindow** is used for transient things, such as drop-down menus and combo boxes. Because our keyboard overlay is not a large, functional part of the app, using a **JWindow** is perfect.

Figure 2 shows what the final app looks like. As I type into any text field or press menu shortcut keys, the most recent key-stroke appears in the overlay window.

Fancy Pop-Down Menus

In the old days of windowing interfaces, we had a fairly small number of UI controls: buttons, sliders, checkboxes,

scroll bars, and menus. That was pretty much it. Today we have many, many kinds of custom controls that behave in lots of different ways.

A popular new kind of control in desktop apps such as Chrome is the pop-down menu (though every platform has its own name for this control). A pop-down menu is a small window containing several actions. It “pops down” from a button on the main interface and is used to expose more functionality to users without taking them away from the main window. Pop-down menus are transient. They appear when the user clicks the button and disappear when the user chooses an action or switches to another window. And, of course, they need to look cool. Sounds like a job for a translucent [JWindow](#).

Figure 3 shows the goal: a pop-down menu with rounded corners containing the actions. It has a pointy tab at the top indicating where the pop-down menu came from, which ensures that the user never gets lost or wonders what the pop-down menu is referring to.

Drawing the pop-down menu itself is pretty easy. The buttons are just regular `JButton` objects in a `JPanel` wrapped in a custom container called the `PopUpTabPanel`. This custom panel sizes itself to fit the child components, but we need some extra space to draw the cool rounded and pointy edges. To do that, I added an empty border using the Swing `BorderFactory`, and then I overrode `paintComponent` to draw the gradient effects.

Listing 6 shows the code.

Notice that I used the Java 2-D [Area](#)

LISTING 6

```
private static class PopupTabComponent extends JComponent {
    public PopupTabComponent(JComponent component) {
        this.setLayout(new BorderLayout());
        this.add(component, BorderLayout.CENTER);
        component.setBackground(new Color(0,0,0,0));
        this.setBorder(BorderFactory.createEmptyBorder(35, 10, 10, 10));
    }
    @Override
    protected void paintComponent(Graphics gfx) {
        Graphics2D g = (Graphics2D) gfx;
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.
VALUE_ANTIALIAS_ON);

        Color[] colors = new Color[]{
            new Color(0.6f,.6f,.6f,0.9f)
            ,new Color(0.9f,0.9f,0.9f,1f)
            ,new Color(0.9f,0.9f,0.9f,1f)
            ,new Color(0.6f,0.6f,0.6f,0.9f)};
        float[] stops = new float[]{0.2f,0.4f,0.9f,1f};
        LinearGradientPaint paint = new LinearGradientPaint(
            new Point(0,0), new Point(0,getHeight()),
            stops,colors);

        //g.setPaint(new Color(255,255,255,120));
        g.setPaint(paint);
        Path2D.Double path = new Path2D.Double();
        path.moveTo(getWidth()/2, 0);
        path.lineTo(getWidth()/2+20, 30);
        path.lineTo(getWidth()/2-20, 30);
        path.closePath();
        Area area = new Area(path);
        RoundRectangle2D.Double rect = new RoundRectangle2D.
Double(0,30,getWidth()-1,getHeight()-30-1,30,30);
        area.add(new Area(rect));

        g.fill(area);
        g.setPaint(new Color(50,50,50,120));
        g.draw(area);
    }
}
```

 [Download all listings in this issue as text](#)



JAVA TECH

ABOUT US

©

f



Java
.ne

blo

39



A 3D rendering of a stylized character. The character has a black, cone-shaped head, a large red sphere for a face, and a white, rounded body. It has black arms and hands, standing with hands on hips. The character is set against a light blue background.



S, and then values can be retrieved from the sequence by using the expression NEXT VALUE FOR S, for example:

```
INSERT INTO CUSTOMERS(ID,NAME)
VALUES (NEXT VALUE FOR S, 'Joe')
```

One of the most frequently requested features for Java DB has been a way to limit the number of rows returned by a query. The latest version of Java DB now supports the `OFFSET/FETCH NEXT` syntax that was introduced in `SQL:2008`.

For example, to select the three highest scores from a table, you would use `FETCH NEXT` like this:

In order to speed up the queries, you can also create an index on the generated column:

And as new people are added to the table, or existing records are updated, the database automatically updates both the generated column and the index. This way, generated columns can be used as a substitute for expression indexes, which are found in some other database systems.

```
SELECT NAME, SCORE FROM RESULTS
ORDER BY SCORE DESC
FETCH NEXT 3 ROWS ONLY
```

And to select the scores ranked from 11 to 20, you would combine `OFFSET` and `FETCH NEXT` like this:

```
SELECT NAME, SCORE FROM RESULTS
ORDER BY SCORE DESC
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
```

Language-based ordering. By default, Java DB orders character strings by doing a character-by-character comparison of

LISTING 3

```
// Define a new type for java.util.List
stmt.execute("CREATE TYPE JAVA_UTIL_LIST " +
    "EXTERNAL NAME 'java.util.List' " +
    "LANGUAGE JAVA");

// Create a table that holds java.util.List objects
stmt.execute("CREATE TABLE T(LST JAVA_UTIL_LIST)");

// Insert lists into the table
try (PreparedStatement ps = conn.prepareStatement(
    "INSERT INTO T(LST) VALUES (?)") {
    ps.setObject(1, Arrays.asList("First element", "Second element"));
    ps.executeUpdate();
    ps.setObject(1, Arrays.asList("A", "list", "with", "five", "elements"));
    ps.executeUpdate();
}

// Retrieve the lists from the database
try (ResultSet rs = stmt.executeQuery("SELECT LST FROM T")) {
    while (rs.next()) {
        List list = (List) rs.getObject(1);
        System.out.println("Size of list: " + list.size());
    }
}
```

 [Download all listings in this issue as text](#)



any type of array, rather than writing one method for `String[]`, one for `Person[]`, and so on. Covariant arrays allow us to write the following:

```
public static boolean find(Object
item, Object[] arr) { ... }
```

Of course, since Java SE 5, we have a better solution. Using generics, the method can be written as follows:

```
public static <T> boolean find(T
item, T[] arr) { ... }
```

This solution is not only guaranteed to be type safe (that is, it will not cause runtime exceptions because of a problem with the array's type), but it has the additional advantage of encoding the constraint such that the type of the item to find and the type of the array must be compatible. However, this solution works only in the presence of generics. Before Java SE 5, using covariant arrays, as shown above, was the only option when implementing a method like this.

Unlike arrays, generics in Java are *invariant*. This protects us from the run-time error in our earlier example involving arrays. For example, the following code will not compile:

```
ArrayList<Fruit> f = new  
ArrayList<Banana>();
```

While this code is type safe, it somewhat limits us in terms of what we can express. Sometimes we want to be able to treat all lists of `Fruit` in the same way,

regardless of whether they are lists of **Bananas** or **Apples**. As long as we do not modify the lists, this should work without causing type errors.

For this reason, Java supports wildcards for generics, which allow us to specify a bound to the instance of a generic. In practice, it can be used to support a form of restricted covariance shown in the following code:

```
ArrayList<? extends Fruit> f = new  
ArrayList<Banana>();  
f.add(new Banana());  
// Prev. line causes compile error
```

In Java, when we use generics with wildcards, as shown above, we then cannot make modifications. For example, in the code above, we can assign an `ArrayList<Banana>` to the variable `f` of type `ArrayList<? extends Fruit>`, but we cannot then add any new items to `f`.

Covariance in Other Languages

Most languages support some form of *variance* of types. Scala, for example, allows developers to annotate a parametric type to indicate whether it should be covariant, contravariant, or invariant (that is, neither covariant nor contravariant). This is called *definition-site variance*.

In the following example, we annotate the parametric type `CoBowl` so that it can be used covariantly (this is indicated by the `+` annotation). Assigning a `CoBowl<Banana>` to `CoBowl<Fruit>` is, therefore, allowed. The parametric `Bowl` type, on the other hand, has no

LISTING 1

```
public class Main
{
    public static void main(String[] args)
    {
        Object[] str = new String[10];
    }
    public static Object[] test2()
    {
        return new String[10];
    }
}
```

 [Download all listings in this issue as text](#)

such annotation and is, thus, invariant. Therefore, a **Bowl**<Banana> cannot be assigned to a **Bowl**<Fruit>:

```
Bowl[T]{} // invariant
CoBowl[+T]{} // covariant as
// indicated by the +
val bowl : Bowl[Fruit] = new Bowl
[Banana]; // compile error
val coBowl : CoBowl[Fruit] = new
CoBowl[Banana]; // correct
```

In order to avoid type errors like the one in our original example, the mutable collections API in Scala was designed to be invariant. However, the immutable collections API can safely be used covariantly.

Some programming languages, such as Google Dart, simply allow covariance in collections in order to avoid cluttering the type system

with additional complexity, despite the fact that this can potentially cause problems at runtime.

Empirical Study of Covariant Arrays

Although the type problems caused by covariant arrays in Java are well known, there has been no study showing how widely this problematic feature is used by developers in practice. We conducted an empirical study on the use of covariant arrays, aiming to determine how

frequently and in which contexts they are used.

We analyzed 64 open source programs from the Qualitas Corpus, ranging from games to system software and libraries. Although the Qualitas Corpus contains 106 programs, we selected only a subset, excluding some of the programs that needed to be fixed in order to compile. A number of

USING WILDCARDS

Java supports **wildcards for generics**, which allow us to specify a bound to the instance of a generic.

This required changing the method `isSubtypeUnchecked(Type t, Type s, Warner warn)`, which is called by the compiler to check whether two types are subtypes (that is, whether or not one type is substitutable for the other). Whenever this

```
Main.java:5: incompatible types
found: java.lang.String[]
required: &java.lang.Object[]
    Object[] = new String[10];
Main.java:10: incompatible types
found: java.lang.String[]
required: java.lang.Object[]
    return new String[10];
```

Looking more closely at the covariant method calls to Java libraries, we found that a small number of library methods accounted for the majority of covariant method calls, as shown in **Figure 3**. The toArray methods of **Vector** and **List** caused 146 covariant calls in our corpus, almost half of all uses of covariant arrays. These methods take an **Object[]** parameter and return an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array. The various methods of **JOptionPane**, which take a parameter of type **Object[]** (for example, **showInputDialog**), were also called covariantly 34 times.

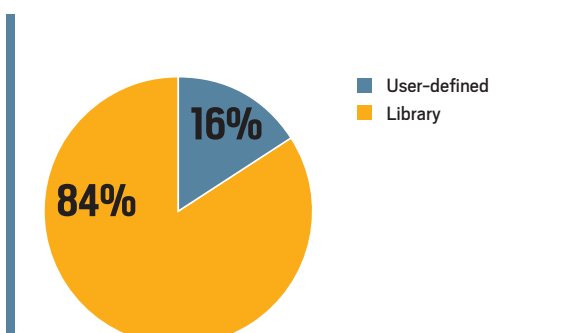


Figure 2



YOUR LOCAL JAVA USER GROUP NEEDS YOU

Find your JUG here



JAVA IN ACTION

JAVA TECH

ABOUT US

○

f



ava
net

log



47

- Types and Programming Languages by Benjamin Pierce (MIT Press, 2002)



Register Now

SAVE \$400

With Early Registration

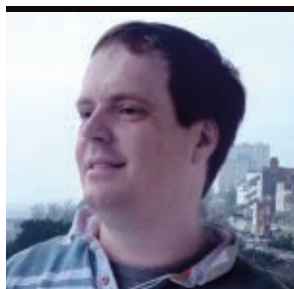
oracle.com/javaone

Bronze Sponsors



Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

ORACLE®



BEN EVANS' PHOTOGRAPH BY
JOHN BLYTHE

Part 1

Introduction to JIT Compilation in Java HotSpot VM

Use the `PrintCompilation` switch to observe the effects of Java HotSpot VM compiling methods during runs.

This article is the first article in a two-part series about Java HotSpot VM and just-in-time (JIT) compilation.

Java HotSpot VM is the VM that Oracle acquired with the Sun acquisition, and it is the VM that forms the basis of both the Java Virtual Machine (JVM) and the open source OpenJDK. Like all VMs, Java HotSpot VM's role is to provide an operating environment for bytecode. In practice, there are three major functions that need to be performed:

- Executing the instructions and computations that are requested by methods
- Locating, loading, and verifying new types (that is, class loading)
- Managing memory on behalf of application code

The last two functions are huge topics in their own right, so in this article we will focus purely on the execution of code.

JIT Compilation

Java HotSpot VM is a *mixed-mode VM*, which means that it starts off interpreting the byte-code, but it can (on a method-by-method basis) compile code into native machine instructions for faster execution.

By passing the switch `-XX:+PrintCompilation`, you can see entries in the log file that show each method as it is compiled.

This compilation takes place at runtime—after the method has already been run a number of times. By waiting until the method is actually being used, Java

HotSpot VM can make sophisticated decisions about how to optimize the code as it compiles the code.

If you're curious about how much difference the JIT makes, you can turn it off using `-Djava.compiler=none` and then look at the difference in your benchmarks.

Java HotSpot VM is capable of running in two separate modes: client or server. You can choose the mode by specifying the **-client** or **-server** switch to the JVM on startup. (This must be the first switch provided on the command line.) Each mode has different situations in which it is usually preferred. In this article, we'll be concerned only with the server mode.

The major difference between the two modes is that the server mode makes more-aggressive optimizations—based on assumptions that might not always hold. These optimiza-

tions are always protected with a simple *guard condition* to check whether the assumption is correct. If, for any reason, an assumption is not valid, Java HotSpot VM reverts the optimization and drops back to interpreted mode. This behavior means that Java HotSpot VM will never do the wrong thing due to an incorrect optimization assumption; it always checks the optimization first.

In server mode, by default, Java HotSpot VM runs a method in interpreted mode 10,000 times before compiling it. You can adjust this value by using the **CompileThreshold** switch. For example, passing **-XX:CompileThreshold=5000** causes Java HotSpot VM to run methods only half as many times before compiling.

It can be tempting for new users to reduce the compile threshold to a very low value. However, you should resist this temptation,

BEST BET

Java HotSpot VM works best when
it can accumulate
enough statistics
to make intelligent
decisions about
what to compile.

tion count (for the number of iterations performed so far).

Note: The `String` and `UTF_8` classes are not used directly by the test, but compilation output still appears for them because they're used by the platform.

On the second line in **Listing 2**, you can see that both tests are very slow. This is because the first run of the code includes the time to load each class. The next line is much faster even though no code tested is compiled at this stage.

Also note the following:

- At 1,000 and 5,000 iterations, direct access to the fields is faster than via getter/setter methods because the getter and setter have not been inlined or even optimized. Even so, both are pretty fast.
- By 9,000 iterations, the getter is optimized (because it is called twice per loop), which gives a slight overall improvement to performance.
- By 10,000 iterations, the setter has been optimized. The extra time spent including the optimized code means the code briefly runs slower.
- Finally, both test classes are optimized:
 - **DFACaller** uses direct access to the fields, and **GetSetCaller** uses the getter and setter. This is the point at which the getter and setter are not just optimized; they are also inlined.
 - You can see that in the next iterations, the test times are still not optimal.
- After 13,000 iterations, the performance for each is as good as the final, much longer test. We've reached steady-state performance.

The important thing to note is that the steady-state performance for accessing fields directly or using getters and setters is basically the same because the methods have (finally) been inlined into `GetSetCaller`, meaning the callable code in `viaGetSet` is doing exactly the same work as the code in `directCall` (which accesses the fields directly).

The JIT compilation is performed in the background, and exactly when each optimization is available for execution varies from machine to machine and, somewhat, from run to run.

Conclusion

In this article, we've shown you the very tip of the JIT compilation iceberg. In particular, we haven't addressed some very important aspects of how to write good benchmarks and how to use statistics to ensure that the dynamic nature of the platform isn't fooling you.

The benchmark used here is very simple, and it isn't suitable for a real benchmark. In Part 2, we plan to show you how to handle a more realistic benchmark and also delve deeper into the code that the JIT compiler produces when it compiles your code. [</article>](#)

LEARN MORE

- Java HotSpot VM
- Just-in-time compilation

LISTING 1

LISTING 1A

LISTING 1B

LISTING 2

```
public class Main {

    private static double timeTestRun(String desc, int runs, Callable<Double> callable)
throws Exception {
        long start = System.nanoTime();
        callable.call();
        long time = System.nanoTime() - start;
        return (double) time / runs;
    }

    // Housekeeping method to provide nice uptime values for us
    private static long uptime() {
        return ManagementFactory.getRuntimeMXBean().getUptime() + 15;
    }
    // fudge factor
    }

    public static void main(String... args) throws Exception {
        int iterations = 0;
        for (int i : new int[]{ 100, 1000, 5000, 9000, 10000, 11000, 13000, 20000,
100000 } ) {
            final int runs = i - iterations;
            iterations += runs;

            // NOTE: We return double (sum of values) from our test cases to
            // prevent aggressive JIT compilation from eliminating the loop in
            // unrealistic ways
            Callable<Double> directCall = new DFACaller(runs);
            Callable<Double> viaGetSet = new GetSetCaller(runs);

            double time1 = timeTestRun("public fields", runs, directCall);
            double time2 = timeTestRun("getter/setter fields", runs, viaGetSet);

            System.out.printf("%7d %,7d\t\tfield access=%.1f ns, getter/setter=%.1f ns%n",
uptime(), iterations, time1, time2);
            // added to improve readability of the output
            Thread.sleep(100);
        }
    }
}
```


[Download all listings in this issue as text](#)



JAMES L. WEAVER



Part 1

Using Properties and Binding in JavaFX 2.0

Implement the power of binding in your JavaFX applications.

JavaFX 2.0, released in October 2011, is an API and runtime for creating Rich Internet Applications (RIAs). One of the advantages of JavaFX 2.0 is that the code can be written in the Java language using mature and familiar tools.

This two-part series focuses on JavaFX 2.0 properties and binding, which are often used to keep the state of the user interface (UI) in sync with the application's model. JavaFX 2.0 comes with a set of interfaces, shown in **Figure 1**, whose purpose is to provide

support for using and implementing properties, detecting when the values of properties have changed, and binding properties to other properties. These interfaces are located in four packages:

`javafx.beans`, `javafx.beans`
`.binding`, `javafx.beans`
`.property`, and `javafx.beans.value`.
This article contains an example
of using the methods defined by
many of these interfaces to use
properties and binding.

Overview of the BindingExampleSolution Application

To help you learn how to use properties and binding, an example application named `BindingExampleSolution` will be employed. As shown in **Figure 2**, this application contains three slider controls and numeric values that you'll keep in sync with the sliders a little later.

GET TOGETHER
Properties
and binding
keep the UI and
application
model in sync.

The BindingExample-Exercise project that you'll download in the next section contains starter code for the example application. In its current form, the application's runtime appearance is similar to **Figure 2**. During the course of this article,

you'll modify the code to implement the binding behavior of the `BindingExampleSolution` application, which is shown in **Figure 3**.

As shown in **Figure 3**, when you move the sliders labeled `InvalidationListener` and `ChangeListener`, the values of the labels to their right reflect the value of the corresponding slider from 0.0 through 100.0.

Also, when you select the Bind checkbox and move the slider labeled bind/unbind, the value to its right reflects the value of the bind/unbind slider added to the product of the value of the top two sliders.



Figure 1

PHOTOGRAPH BY
STEVE GRUBMAN

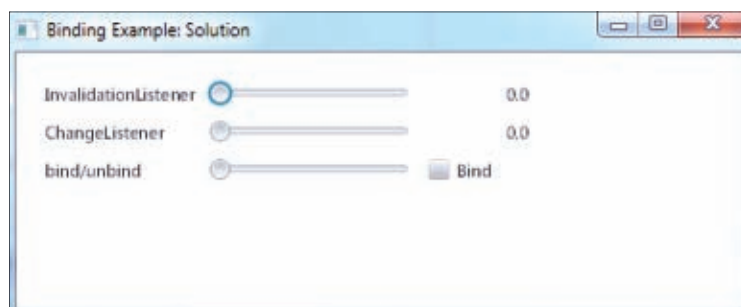


Figure 2

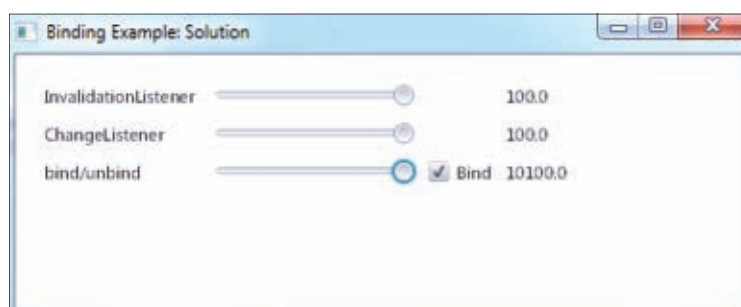


Figure 3

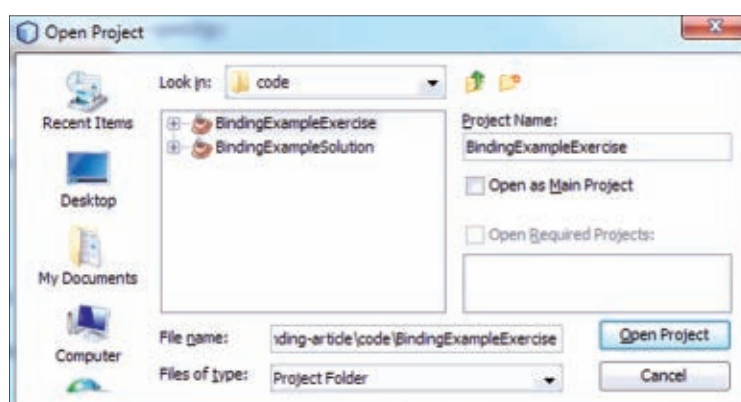


Figure 4



Figure 5

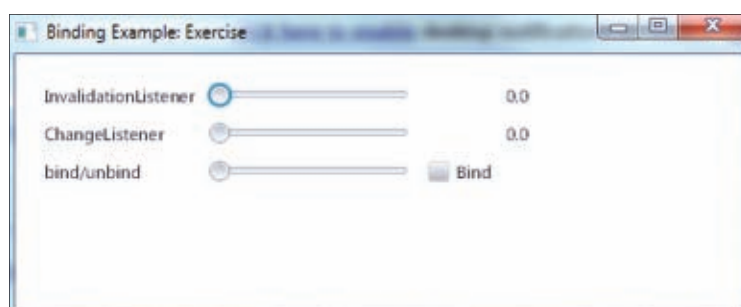


Figure 6

Obtaining and Running the BindingExampleExercise Project

1. Download the [NetBeans project file](#), which includes the BindingExampleExercise project.
2. Expand the project into a directory of your choice.
3. Start NetBeans, and select **File -> Open Project**.
4. From the Open Project dialog box, navigate to your chosen directory and open the BindingExampleExercise project, as shown in **Figure 4**. If you receive a message stating that the jfxrt.jar file can't be found, click the **Resolve** button and navigate to the rt/lib folder subordinate to where you installed the JavaFX 2.0 SDK.

Note: You can obtain the NetBeans IDE from the NetBeans site.

5. To run the application, click the **Run Project** icon on the toolbar, or press the **F6** key. The Run Project icon looks like the Play button on a DVD player, as shown in **Figure 5**.

The BindingExample-Exercise application should appear in a window, as shown in **Figure 6**.

LISTING 1

```
final DoubleProperty valueA = new SimpleDoubleProperty(0);

Slider sliderA = SliderBuilder.create()
    .max(100)
    .build();
sliderA.valueProperty().bindBidirectional(valueA);
```



[Download all listings in this issue as text](#)

Notice that moving the sliders has no effect. Your mission will be to add code that implements the behavior described previously. The next sections describe the steps you can follow to implement this behavior.

Step 1: Adding an InvalidationListener to Detect Changes to the Value of the Top Slider

To dynamically update the label text as the top slider is moved, we'll add an `InvalidationListener` to a property that tracks with the value of the slider.

Take a look at the code in `BindingExampleMain.java` in the `BindingExampleExercise` project, which shows the starter code for this example. We'll show code snippets from `BindingExampleMain.java` as you perform the steps in this exercise.

Instantiate a SimpleDoubleProperty that tracks with the value of the slider. The starter code in `BindingExampleMain.java` instantiates a `SimpleDoubleProperty` and bidirectionally binds the value property of the slider with it, as shown in **Listing 1**.

The `SimpleDoubleProperty` class is one of several classes (for example, `SimpleBooleanProperty`) in the `javafx`

`.beans.property` package that can create properties that represent most of the Java datatypes.

The value property returned by the `valueProperty()` method of the `Slider` class is of type `DoubleProperty`. `DoubleProperty` is an abstract class that implements the `Property` interface shown in the Unified Modeling Language (UML) diagram in **Figure 1**, which contains a `bindBidirectional()` method.

We're using this `bindBidirectional()` method to bidirectionally bind the value property of the slider to the `DoubleProperty` referenced by the `valueA` variable. As a result, when the user moves the slider, the value of the `DoubleProperty` referenced by `valueA` will track with the value property of the slider. Conversely, if the value of the `DoubleProperty` referenced by the `valueA` variable is changed, the value property of the slider will change as well.

Add an `InvalidationListener` that updates the text property of the label. As shown in the UML diagram in **Figure 1**, the `Property` class indirectly extends the `Observable` interface. The code shown in **Listing 2** from `BindingExampleMain.java` leverages the `addListener()` method

defined in the `Observable` interface to add an `InvalidationListener` to the `DoubleProperty` referenced by `valueA`.

This `invalidated()` method is invoked whenever the `valueA` property is no longer valid, which means that the value *might* have changed. To update the label to the right of the slider, we'll put some code in the `invalidated()` method that casts the `Observable` object to a `DoubleProperty`, gets its value, converts it to a string, and sets the text of `labelA` to the string.

Go ahead and fill in the lines indicated by the “TO DO” comments, so the code in **Listing 2** turns into the code shown in **Listing 3**.

Note: This use of an `InvalidationListener` is for demonstration purposes, showing that the `Observable` argument passed into the `invalidated()` method can be read with the `getValue()` method. The act of reading the value, however, forces it to be reevaluated, because the implementation of `SimpleDoubleProperty` uses *lazy evaluation*. This result defeats the

purpose of using an **InvalidationListener**, which is to be notified that a value is no longer valid without forcing it to be reevaluated. In contrast, the purpose of the **ChangeListener** (which you'll experience next) is to read the value as soon as it changes.


Now that you've implemented the `invalidated()` method, run the application to try the slider labeled `InvalidationListener`. After doing so, let's address the middle slider, which is labeled `ChangeListener`.

Step 2: Adding a ChangeListener to Detect Changes to the Value Property of the Middle Slider

To dynamically update the label text as the middle slider is moved, this time we'll add a **ChangeListener** to the property that tracks with the value of the middle slider.


As in Step 1, the starter code for the middle slider in `BindingExampleMain.java` instantiates a `SimpleDoubleProperty` and



 **Jim Weaver introduces the concept of binding in JavaFX 2.0.**

LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6 / LISTING 7

```
valueA.addListener(new InvalidationListener() {
    public void invalidated(Observable observable) {
        // observable is a DoubleProperty
        // TO DO: Cast the Observable object to a DoubleProperty; get its value
        //      and convert it to a String, and set the text of
        //      labelA to the String.
    }
});
```

 [Download all listings in this issue as text](#)

bidirectionally binds the value property of the slider with it, as shown in **Listing 4**.

Add a `ChangeListener` that updates the text property of the label. As shown in the UML diagram in **Figure 1**, the `Property` class indirectly extends the `ObservableValue` interface. The code shown in **Listing 5** from `BindingExampleMain.java` leverages the `addListener()` method defined in the `ObservableValue` interface to add a `ChangeListener` to the `DoubleProperty` referenced by `valueB`.

This `changed()` method is invoked whenever the `valueB` property has changed. To update the label to the right of the slider, we'll put some code in the `changed()` method that converts the `newValue` to a string and sets the value of the text property of `labelB` to the string.

Go ahead and fill in the lines indicated by the “TO DO” comments, so the code in **Listing 5** turns into the code shown in **Listing 6**.

You might recall that in **Listing 3**, we used the `setText()` method of `labelA` to set its text property. To demonstrate an

alternative, here we're using the `textProperty()` method of `labelB` to get the underlying `StringProperty` and we're using its `setValue()` method to set the text.

Now that you've implemented the `changed()` method, run the application again to try the slider labeled `ChangeListener`. After doing so, let's address the bottom slider, which is labeled `bind/unbind`.

Step 3: Using bind(), unbind(), and Bind Expressions to Manage the Text Property of the Bottom Label

To dynamically update the label text as the bottom slider is moved, this time we'll bind the text property of the label to the properties that track with the values of the sliders.

As in Step 2, the starter code for the bottom slider in `BindingExampleMain.java` instantiates a `SimpleDoubleProperty` and bidirectionally binds the value property of the slider with it, as shown in **Listing 7**.

Add a `ChangeListener` to the checkbox.

As shown in **Figure 3**, there is a checkbox labeled Bind that, when selected, causes

Detecting when the status of the checkbox changes is handled by the code in **Listing 8** from [BindingExampleMain.java](#).

Create bind expressions. The bind (actually, the bidirectional bind) from **Listing 7** binds one property to another property of the same type, with both properties holding the same value. It is often desirable to bind an expression that contains one or more properties to

To demonstrate this technique, we'll put some code that uses a *bind expression* into the `changed()` method from **Listing 8**.

SERIOUS SUPPORT

JavaFX 2.0 comes with **a set of interfaces whose purpose is to provide** support for implementing properties, detecting when the values of properties have changed, and binding properties to other properties.

```
checkboxC.selectedProperty().addListener(new ChangeListener<Boolean>() {
    public void changed(ObservableValue ov,
        Boolean oldValue, Boolean newValue) {
        if (newValue.booleanValue()) {
            // TO DO: Using the bind() method and the Fluent API, bind the text
            // property of labelC to the product of valueA and valueB plus valueC
        }
        else {
            labelC.textProperty().unbind();
        }
    }
});
```

the bottom slider and the Bind checkbox exhibit the expected behavior.

As shown in **Listing 9**, a bind expression consists of methods chained together that comprise the expression. These methods are inherited from classes that exist in the `javafx.beans.binding` package and, in this case, they are inherited by the `DoubleProperty` from the `DoubleExpression` and `NumberExpressionBase` classes.

Bind expressions are sometimes referred to as using the *fluent interface API*, which is a programming style in which a method chain reads like a fluent sentence.

Also shown in **Listing 9** is an `unbind()` method invocation, which breaks the bind that was previously established with the `bind` expression.

JavaFX 2.0 comes with numerous classes and interfaces that provide a

powerful properties and bindings framework. This framework fires two kinds of events—invalidation events and change events—which are handled by the `InvalidationListener` and `ChangeListener` interfaces, respectively.

Properties can be bound to each other, either unidirectionally or bidirectionally. You can also leverage bind expressions to bind the value of one property to an expression that contains one or more properties.

In Part 2, we'll look at optimizing JavaFX 2.0 properties and bindings by implementing lazy initializations and custom bindings. </article>

- [The properties and binding tutorial](#) at Oracle's JavaFX site
- [Michael Heinrichs' blog](#), which includes entries on JavaFX properties and bindings

Find your JSR here





JAVA TECH

ABOUT US

We have the current location data, so next, we need to establish a separate **Canvas** class for displaying the viewfinder and taking photos.

This is where we intervene to modify the bytecode that is gen-

0

ava
net

log



erated to add the location data. To do so, we first need to understand the file structure, because we will be modifying it at the byte level.

The EXIF Format
EXIF stands for *exchangeable image file format*. In spite of its name, it was originally created to tag any media with metadata information. It is based on the JPEG file interchange format (JFIF) and, more importantly, you can insert EXIF format data into existing JPEG images to tag them with the correct metadata.

You can think of the EXIF format as a special container within a JPEG file for holding metadata for that image. This container is also capable of holding thumbnail information so that the original JPEG image becomes more useful for display purposes.

You can easily identify a JPEG file that holds an EXIF metadata container by opening such a file within a Hex viewer and looking for the text *Exif*. The data that follows this text is what defines the EXIF structure.

NOT JUST IMAGES
EXIF was originally created to tag any media with metadata information.

However, even before you get to the *Exif* text, you will see items that the JPEG standard defines as *application markers*. These markers start with the special bytes **0xFFE0** through **0xFFEF**. While the application markers are not necessary for rendering the image data, they provide processing information. EXIF uses the special **0xFFE1** marker to indicate that the EXIF application marker is starting.

Following the **0xFFE1** marker, the length of the marker is specified using 4 bytes (including the marker). Finally, there is the special string *Exif*, which marks the start of the EXIF structure.

So far, we have the following structure for an EXIF encoded metadata image file, where **LLLL** is the total length of the EXIF application marker, **0x45** through **0x66** represent the string *Exif*, and the last two **0x00** bytes mark the end of this section:


0xFFE1 LLLL 0x45 0x78 0x69 0x66 0x00 0x00

DESCRIPTION OF BYTES	VALUE OF BYTES
4 BYTES THAT TELL US HOW MANY DIRECTORY ENTRIES ARE IN THIS IFD:	0X0003
THE FIRST DIRECTORY ENTRY:	TTTTFFFFNNNNNNNDDDDDDDD
THE SECOND DIRECTORY ENTRY:	TTTTFFFFNNNNNNNDDDDDDDD
THE THIRD AND FINAL DIRECTORY ENTRY:	TTTTFFFFNNNNNNNDDDDDDDD
8 BYTES THAT SHOW THE OFFSET TO THE NEXT IFD (OR 0X0000, IF THIS IS THE LAST IFD):	0X00000000

Table 1

LISTING 1 / LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5

```
Criteria criteria = new Criteria();
criteria.setHorizontalAccuracy(Criteria.NO_REQUIREMENT);
criteria.setVerticalAccuracy(Criteria.NO_REQUIREMENT);
criteria.setPreferredResponseTime(Criteria.NO_REQUIREMENT);
criteria.setCostAllowed(false);
```

 [Download all listings in this issue as text](#)

Following this, there is a mandatory TIFF header structure:

■ **0x49 0x49 0x2A 0x00**

This leads to the next 4 bytes, which tell us where in this file structure we will find a directory of information for the image data. These 4 bytes are like an offset to the actual data. This data, which has a special structure, is called an image file directory (IFD).

Understanding the IFD

The last 4 bytes of the TIFF header are a pointer to the actual location of the first IFD. At the end of the first IFD, the last

4 bytes then point to the location of the next IFD, and so on until there are no more IFDs. The last IFD has **0x0000** as the last 4 bytes to signify that there are no more IFDs.

The IFD is like a compact directory of information. Each EXIF container can hold multiple directories that form a chain, with the first one pointing to the next through its last 4 bytes.

We are mainly interested in the GPS IFD because we want to insert our GPS metadata there, but all IFDs have the same structure, as shown in **Table 1**.

In **Table 1**, the first 4 bytes state that there are three directory entries within this IFD. Next are the three directory

entries, each with a structure that provides the data the directories hold. Finally, the 8 bytes at the end link to the next IFD, if there is one, or they contain all zeros, if there are no more IFDs. Typically, this would mark the end of the EXIF container as well.

Each directory entry follows the same structure. The first 4 bytes (TTTT) represent a tag number that is relevant to the type of IFD. Because we are concentrating on GPS IFDs, the first tag would be 0x0000, which represents the GPS version number. Thus, what we are saying is that this directory entry contains the GPS version number.

The next 4 bytes (FFFF) tell us the format of this data. There are 12 different types of data formats, from ASCII, to byte, to double float. Each type has its own number, for example, ASCII is represented by the number 2 and unsigned rational is represented by the number 5. Thus, if this were the directory structure for the GPS version number, these 4 bytes would contain the value 0x02 0x00.

The next 8 bytes are **NNNNNNNN**, and they tell us how many components are available for this value. Each data value might have multiple components that together make up the data. Because the GPS version number contains only a single component, we would just use the value **0x01 0x00 0x00 0x00**.

Finally, the last 8 bytes
(DDDDDDDD) are where the

data is stored or they are the offset to where the data is stored. For the GPS version number, this is typically **0x32 0x32 0x00 0x00** (representing GPS version 2.2.0.0).

With this understanding of the EXIF file structure behind us, let's move on to the actual code.

Modifying the Image Data

In the prior sections, we gathered the location data and took a snapshot. We will gather this data in a byte array and proceed to modify it, as shown in **Listing 6**.

I haven't shown it here, but we are doing this process (including the taking of the snapshot) in a separate thread so we don't block the main application. (See the complete source code.)

Now we will go through this array one byte at a time and modify it accordingly, as shown in **Listing 7**.

The EXIF format uses the marker `0xE1`. The image that we process using the Sun Java Wireless Toolkit has the marker `0xE0` to represent a pure JPEG image. We replace that with the `E1` tag and then write the total size (which we calculate manually).

Then we write the familiar *Exif* string to identify that we are now writing an EXIF section. This is followed by the mandatory TIFF header. See **Listing 8**.

We then have the first IFD, the GPS IFD, as shown in **Listing 9**. Note that the GPS

WRITE AN APP

Geotagging is **not enabled by default in GPS-enabled devices**, so you need to write your own special application to add the metadata.

LISTING 6 / LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10

```
imageArray = videoControl.getSnapshot("encoding=jpeg");
// read it
ByteArrayInputStream bis = new ByteArrayInputStream(imageArray);
// for writing the modified array
ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

 [Download all listings in this issue as text](#)

IFD doesn't contain the actual data, but an offset to the location of the actual GPS IFD structure, which starts with an indication of how many entries it contains (three: version, latitude, and longitude) followed by the first entry (the GPS version), as shown in **Listing 10**.

The next entry is the GPS latitude. See **Listing 11**. Notice that it doesn't actually

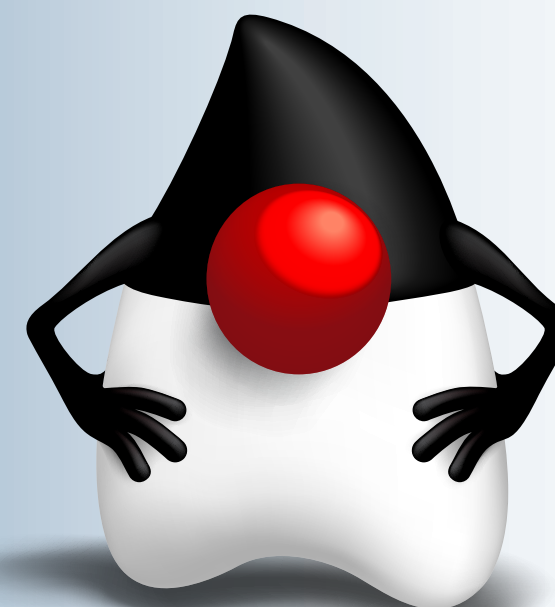
contain the data. It links to the data via the offset.

I will skip the GPS longitude entries to show you the offset data (see **Listing 12**).

I used a special method to convert the GPS latitude to degrees and minutes from a decimal representation. The format expects the degrees and minutes that make up the GPS informa-

YOUR LOCAL JAVA USER GROUP NEEDS YOU

Find your JUG here



```
// second entry is GPS Latitude - 2 bytes
bos.write(0x02);
bos.write(0x00);

// data format (rational) - 2 bytes
bos.write(0x05);
bos.write(0x00);

// number of components (2 - degrees and minutes) - 4 bytes
bos.write(0x02);
bos.write(0x00);
bos.write(0x00);
bos.write(0x00);

// offset to data value (64) - 4 bytes
bos.write(0x40);
bos.write(0x00);
bos.write(0x00);
bos.write(0x00);
```



Download all listings in this issue as text

only briefly on the Location API and taking snapshots. We then discussed modifying the image data to hold EXIF metadata information, covered the basics of EXIF, and worked on the code for an actual application. </article>

- "Getting Started with Touchscreen UIs in Java Platform, Micro Edition"
- "Working with the XML Parser API — JSR 172"
- "Working with the Mobile Sensor API"

This article combined three separate technologies to modify an image to hold GPS metadata information. We touched



JAVA TECH

ABOUT US

0

f

ava
net

log



59

NOT SO OBVIOUS
Convention over
Configuration
in Java EE 6
**requires you to
specify only the
unconventional.**
Obvious choices
do not have to be
configured.

ated and returned. To distinguish a staged injection from the “usual” injection, the `connect` method is annotated with the `@StageDependent` qualifier (see **Listing 4**).

A `Qualifier` “marker” has no attributes and is used only to extend the type of the injected object (see **Listing 5**).

The user of **EISConnector** only has to use the

PHOTOGRAPH BY
THOMAS EINBERGER/
GETTY IMAGES

`@RequestScoped` and `@Named` annotations. On each request, a new instance of the `EISConnector` is injected to the `Index` managed bean. A stateless session bean would cause only a single injection per pooled instance.

Configuration with Convention

In addition to stages, any arbitrary primitive (String, int, long, float) and other objects can be configured dynamically. Let's assume that an imaginary customer expects us to build a highly configurable **Greeter** component (see **Listing 6**).

The **Greeter** stateless session bean returns a configurable message that is concatenated a configurable number of times. We simply use the **@Inject** annotation to inject primitive members, as shown in **Listing 6**.

The injection of the `Greeter#message` and `Greeter#repetition` attributes immediately causes the deployment error shown in **Listing 7**.

The “unsatisfied dependencies” error indicates the lack of adequate instances for the injection. We have to make the primitive available for injection first by using the `@Produces` annotation. The easiest possible solution for the “unsatisfied dependencies” problem is direct exposure of attributes with matching types in a dedicated class:

```
public class Configurator{
    private String msg =
"configurable";
    private int repetition = 2;
}
```

Although the introduction of the **Configurator** class fixes the problem for our simplistic demands and centralizes all configuration data in one place, this approach is not usable in real projects. You could configure only a single String that way. The Contexts and Dependency Injection (CDI) runtime matches the attribute types, not the names. Regardless of the attribute's name, all Strings would get the same value produced in the **Configurator**.

Any attempt to produce an additional String would result in exactly the opposite problem: an “Ambiguous dependencies” error. A custom `@Qualifier` would fix the ambiguity problem, but that approach would require you to write a custom `@Qualifier` annotation for each injected configurable type.

The built-in `@Qualifier javax.inject.Named` uses a String for disambiguation of different injection points, but that introduces another problem. A String-based dependency solution becomes hard to refactor and is, therefore, hard to maintain. The [JSR 299](#) specification states the following in Chapter 3.11:

"The use of `@Named` as an injection point qualifier is not recommended, except in the case of integration with legacy code that uses string-based names to identify beans."

Instead of introducing explicit qualifiers, we could rely on convention and derive the configuration key from

LISTING 1

```
@Model
public class FacesContextExposer {
    @Produces
    public Stage stage(){
        FacesContext ctx = FacesContext.getCurrentInstance();
        return Stage.valueOf(ctx.getApplication().getProjectStage().name());
    }
}
```

 [Download all listings in this issue as text](#)

the injection point's name. The `javax.enterprise.inject.spi.InjectionPoint` instance represents the injection point and is available for a producer method.

With the necessary type name of the attribute with its declaring class extracted from the `InjectionPoint`, we could dynamically configure the whole system (see **Listing 8**).

Now the name of the injection point is used to fetch the corresponding value from a given configuration store. A simple attribute's name might not be unique enough in a larger project, so you could prepend the fully qualified class name to the attribute's name to make it globally unique. The fully qualified class name can be

obtained from the `InjectionPoint` as well (see **Listing 9**).

The conventional use of the type information for attribute configuration requires the fully qualified names of the attributes to be maintained in the configuration store. A dependency between the configuration and the fully qualified class names can cause inconsistencies if the injection point changes its name or the enclosing class moves to another package. For this reason, an additional annotation is introduced to set the name of the configuration key explicitly (see **Listing 10**).

`Configurable` is a Plain Old Java Annotation with a single attribute, `String value()`, and is fully independent of the Java EE API. Because `Configurable` is not denoted with any CDI meta-annotation, such as `@Qualifier` or `@Stereotype`, the application server ignores it.

WATCH OUT FOR THIS
A dependency
between the
configuration and the
fully qualified class
names can cause
inconsistencies.

We are using the `Configurable` annotation to override the configuration key. If a field is denoted with `Configurable`, we are using the `value()` as a lookup key instead of using the fully qualified field name (see **Listing 11**).

The configuration consumer only has to provide the desired lookup key by denoting the field with the `@Configurable` annotation and providing the `value()`:

```
@Stateless
public class Greeter {
    @Inject @Configurable
    ("greetings")
    private String message;
    //..
}
```

Stage-Dependent Configuration

The injection of primitives and the staging mechanism can be combined easily. The already introduced `@StageDependent` qualifier can be used to produce stage-dependent configuration entries. Also, the exposed `Stage` is injected to the

`getString` method and used to build a stage-dependent key, as shown in **Listing 12**.

With the prepended stage name, the field name or the value of the `@Configurable` annotation is used to fetch the configuration value. Now we are not only able to inject different implementations of an interface, but we

can also provide stage-dependent configuration via injection of primitive types:

```
@Stateless
public class Greeter {
    @Inject @StageDependent
    private String stagedMessage;
}
```

The consumer only has to mark the injected field as `@StageDependent` to get environment-dependent values.

Conventional Configuration Store

The centralization of code dealing with configuration is the main benefit of the approach discussed here. All configuration requests are served from a single place: the `Configurator` class. We could apply the “Convention over Configuration” spirit to the `Configurator` also and provide the default values for all obvious configuration entries out of the box (see **Listing 13**).

In the `fetchConfiguration` method of the `@Singleton Configurator`, a `java.util.HashMap` is populated with default values. In our sample, a single `Map` is used for all stages, but it could be split easily into dedicated `Map` instances for each stage. Centralizing suitable defaults in a central `Map` simplifies maintenance. To change the values, however, you will have to recompile the `Configurator` class, which might not be desired for volatile entries.

Plug-ins (or the Sky Is the Limit)

By encapsulating the configuration population with the `ConfigurationProvider`

LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11 / LISTING 12

```
org.jboss.weld.exceptions.DeploymentException: org.jboss.weld.exceptions.  
DeploymentException: WELD-001408 Unsatisfied dependencies for type [String]  
with qualifiers [@Default] at injection point [[field] @Inject private com.abien.  
configuration.business.primitives.consumer.Messenger.message] [...]
```

 [Download all listings in this issue as text](#)

interface, we introduce a flexible extension mechanism:

```
public interface
ConfigurationProvider {
    public Map<String,String>
getConfiguration();
}
```

The `ConfigurationProvider` interface is injected to the `Configurator` wrapped with the `javax.enterprise.inject.Instance<ConfigurationProvider>` interface. With `javax.enterprise.inject.Instance<ConfigurationProvider>`, the injection of the `ConfigurationProvider` interface is lazy and will work even if no valid implementation of the interface is deployed (see **Listing 14**).

Interestingly, the `javax.enterprise.inject.Instance` inherits from the `java.lang.Iterable` interface, which allows iteration over all implementations in an ordinary loop. We loop over all found implementations (see the `Configurator#mergeWithCustomConfiguration`), fetch the external

`Map<String,String>` instance, and merge it with the default values hardcoded in the `@PostConstruct` method. It is the simplest possible implementation of a plug-in mechanism in Java EE: you can easily extend existing functionality with external implementation on demand.

Because we already relied on JSF to provide the staging information, we could also use JSF as a configuration provider. We only have to expose the Servlet's `contextParams` as a `Map<String,String>` in the `FacesContextExposer` (see **Listing 15**).

In the next step, we implement the `ConfigurationProvider` interface and inject it with the `FacesContextExposer` provided by the `Map` instance (see **Listing 16**).

Now the hardcoded default values can be overridden easily in web.xml by `context-param` entries (see **Listing 17**).

You are not limited to web.xml. Other implementations of the **ConfigurationProvider** interface could fetch the implementation from XML, property files, or databases with the Java

JAVA IN ACTION

JAVA TECH

ABOUT US

○

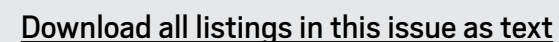
f

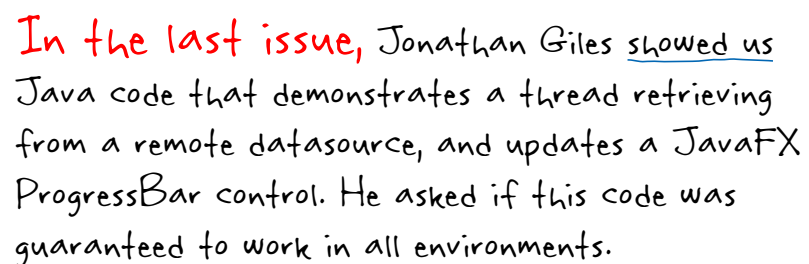
ava
net

olog



62





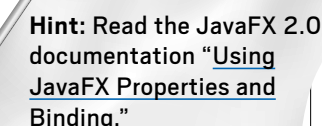
This issue's challenge comes from Angela Caicedo, a Java evangelist at Oracle.

One of the coolest features of JavaFX is binding capabilities. When JavaFX properties participate in bindings, changes made to one object are automatically reflected in another object. This is particularly useful when creating graphical user interfaces because you can automatically keep the display synchronized with the underlying data.

Consider the following code fragment for a JavaFX application:

What will be the output when you try to run this code?

- 1) My bound integer's value is: 8
My bound integer's value is: 5
- 2) My bound integer's value is: 5
My bound integer's value is: 5
- 3) My bound integer's value is: 8
My bound integer's value is: 8
- 4) My bound integer's value is: 8
Exception***



Look for the answer in the next issue. Or submit your own code challenge!

